REMARKS

This Supplemental Response cancels Claims 20-25 to reduce the number of issues, amends Claims 1, 2, 4 and 16-19, and adds Claims 26-32 to better define the invention.

Withdrawal of the rejection of claims under 35 USC §112 and 35 USC §101 is requested for the reasons set forth in the previous Response. Further, withdrawal of the objection to the specification is requested for the reasons set forth in the previous Response.

With regard to the claims, the previous comments with respect to the prior art is no longer considered pertinent in most aspects in view of the claim amendments and the following discussion.

The rejection of Claims 1-5, 10, 11, 13-15 and 17-19 under 35 USC §102(b) as being anticipated by U.S. Patent No. 6 058 460 to Nakhimovsky has been considered.

Applicant's invention is directed to a method for regulating access to data in a data storage device and to a data storage device. In Applicant's method, specific address statements point to specific locations in the data storage device where information for a specific individual system is stored. Thus, a single address statement including the address and length thereof can access a pre-selected amount of data stored in the storage device without requiring additional reservation requests to obtain fragmented data from other memory locations.

The invention dynamically reserves portions, including expansion areas, of a continuous data storage memory for an individual system. A second individual system having a nearby address can reserve the expansion area from the first individual system or utilize portions of the expansion area that is not currently used by the first system. In this way, memory in the data storage device that is not utilized by one individual system may be provided at least temporarily to an adjacent block of memory corresponding to a different individual system. Thus, instead of a fixed block of reserved

memory in a data storage device corresponding to each
individual system, Applicant's invention provides each
individual system with a dynamic amount of memory by providing
speculatively extended areas that include directly required
areas and expansion areas.

Nakhimovsky discloses memory allocation in a multi-
threaded environment. The system establishes memory pools 38,
39, 40 in a system memory for each thread. For each thread of
the multi-threaded computer arrangement, dynamically allocated
user memory blocks from the associated memory pool are
provided. The disclosed method allows any existing memory
management package to be converted to a multi-threaded version
so that multi-threaded processes are run with greater
efficiency. As disclosed at column 1, lines 54-57 of
Nakhimovsky, each thread uses memory allocation routines
(malloc) to manipulate its own memory in its own memory pool.

In Nakhimovsky, the size of a memory pool can be
increased by allocating additional memory from a system
memory. The memory, however, does not correspond to a
specific area in the system memory 14, but instead is taken
from a location of free memory with a previously unknown
address. Column 6, lines 45-55 of Nakhimovsky discloses that
as memory blocks are repeatedly allocated, freed and
reallocated by a thread, the memory pool may become fragmented
into smaller and smaller blocks. This method differs from
Applicant's claimed system wherein each individual system is
provided with an address statement pointing to an exact
location in the data storage device and in use sends and
retrieves data from that particular storage or memory area
including adjacent contiguous expansion areas within the data
storage device.

Column 6, lines 56-67 and column 7, lines 1-25 of
Nakhimovsky disclose the use of a malloc or realloc system and
then later merging malloc pools. A malloc routine as
discussed on the attached Wikipedia entry for "malloc"
operates in a main fixed memory by utilizing a memory stack.

As discussed therein a "heap" of memory is provided and a
pointer simply stacks memory blocks until the program
terminates or a memory is explicitly deallocated. As
discussed on page 5 of the entry under the heading "Heap-
based", fragmentation can occur in such a system and thus a
memory pool must be deployed. The heap requires enough system
memory to fill its entire length, and its base can never
change. Thus, any large areas of unused memory are wasted.
Thus, any large areas of unused memory can be wasted. If a
small used segment exists at an end of the heap, the small
used segment could waste any magnitude of address space.

Applicant's claimed method and system differ greatly from
Nakhimovsky. Claim 1 recites that "each said individual
system reserves free data areas or address areas of a
speculatively extended area in the data storage device and
blocks the reserved areas from access by other said individual
systems, wherein the reserved areas are speculatively extended
by reserving expansion areas". Thus, Applicant's claimed
invention provides each individual system with a speculatively
extended area. The individual system reserves directly
required areas therein and can also reserve expansion areas of
the speculatively extended areas. Thus, when additional data
is received for an individual system, the data goes into an
area specified for the individual system and the overall size
of the reserved area may change as necessary. Further, the
reserved areas are blocked from other individual systems.
Speculatively extended areas that include directly required
areas and expansion areas as recited in Claim 1 are not
believed present in Nakhimovsky.

Applicant's Claim 2 recites that the individual systems
"identify the directly required area from at least one address
statement". Thus, the claimed invention has an address
statement pointing to a specific area in the data storage
device. Nakhimovsky utilizes the malloc system having a heap
system and thus does not have a specific address statement
assigned to the data of a specific individual system. Instead

data is moved freely generally in sequence and stacked by the heap arrangement.

Applicant's Claim 4 recites that "two of said individual systems have expansion areas that overlap each other to provide a common area in the data storage device for access only by the two said individual systems". The expansion areas that overlap with each other physically overlap in the data storage device and thus share an address. Nakhimovsky does not disclose only two individual systems sharing a common area.

Independent Claim 14 recites a data storage device for regulating access to data in a system comprising a plurality of individual systems. Claim 14 recites a reservation means for reserving at least one of free data areas and address areas as reserved areas of a speculatively extended area in the data storage device. The respective reservation means block the reserved areas from access by other said individual systems, and "the reservation means are capable of selectively reserving expansion areas of said speculatively extended areas along with reserving directly required areas of said speculatively extending areas". As discussed above, Nakhimovsky discloses memory pools and a malloc method of distributing data. There is no disclosure of a speculatively extended area for an individual system, the speculatively extended area having expansion areas that can be reserved to vary the reserved area for an individual system. In Nakhimovsky memory pools and a mammoc stack and pointer are provided.

Applicant's Claim 15 further recites a directly required area that is directly required by the individual system by "evaluating at least one address statement provided in a reservation request requesting the directly required area". As discussed above, Nakhimovsky does not disclose an address statement to request a directly required area. In Nakhimovsky, the data locations in the memory change in the malloc operation so that there is no specific directly

required area, much less an address statement that points to that specific directly required area in the system memory.

Claim 17 specifies a plurality of individual systems having requesting means and that "the data storage device blocks the reserved areas from access to other said individual systems" and that the requesting means are "capable of reserving at least portions of the expansion areas of the speculatively extended area that are speculatively extended along with the directly required areas". As discussed above, Nakhimovsky does not disclose this type of arrangement.

Claim 19 recites that "two said individual systems respectively identify and are capable of accessing the common memory area from at least one address statement". As discussed above, Nakhimovsky does not disclose or suggest the claimed arrangement wherein there is a common area for two individual systems. Instead, each individual system can access allocated areas in the memory of other systems that are not blocked.

For the above reasons, reconsideration and allowance of Claims 1-5, 10, 11, 13-15 and 17-19 is respectfully requested.

Claims 20-22 have been cancelled and thus the issues related thereto are moot.

The rejection of Claims 6-9, 12 and 16 under 35 USC §103 as being unpatentable over Nakhimovsky in view of U.S. Patent No. 5 826 082 to Bishop has been considered.

Dependent Claims 6-9, 12 and 16 are believed allowable for the reasons set forth above with regard to the respective parent independent Claims 1 and 14.

Bishop discloses a method for reserving resources in a computer system that includes a resource status table 125, reservation table 130 and in-use table 145. The memory is utilized by a plurality of thread to execute programs thereon.

As illustrated in Figure 1 and disclosed at column 2, lines 58-65, Bishop utilizes a stack arrangement 140 for each thread. Further, a stack pointer and program counter, etc. is provided for each thread.

Bishop does not address the deficiencies of Nakhimovsky with regard to the independent claims. Further, Bishop does not disclose specific expansion areas for individual systems, but instead utilizes a stack arrangement that is comparable to the malloc arrangement of Nakhimovsky.

Applicant's Claim 6 recites a reserved expansion area being released "upon a reservation request relating to at least part of the reserved expansion area from another said individual system or from an additional status storage device". Thus, one individual system can request an expansion area from an adjacent system in some instances and obtain release thereof. As discussed above, Nakhimovsky relies on generally fixed memory pools for each system and a malloc pointer and stack arrangement. Bishop does not disclose or suggest this feature.

Applicant's Claim 7 recites that the expansion area from the individual system is released upon a reservation request coming from another said individual system "if said expansion area that is requested is a directly required area for said another individual system". Thus, in some instances a portion of a reserved expansion area is released. This type of arrangement is not disclosed or suggested in Nakhimovsky. Bishop discloses stacks for the corresponding threads that operate in a stack arrangement, similar to malloc as disclosed in Nakhimovsky. Thus, Bishop does not disclose or suggest releasing, a reservation request for an expansion area of one individual system to another individual system.

Applicant's Claim 8 recites that the expansion area is also released upon a reservation request coming from another individual system "if said expansion area that is requested is an expansion area for another individual system". Bishop does not disclose expansion areas for individual systems that interact to share specific expansion areas. Instead, Bishop relies on a stack method as shown by stacks 140 in Figure 1. As discussed above, Nakhimovsky also relies on a malloc stack method.

Applicant's Claim 9 recites that "only a particular part of the expansion area is released upon a reservation request coming from another said individual system if said expansion area likewise relates only to the expansion area of said another individual system". Again, Bishop does not disclose an arrangement having directly required areas and expansion areas, much less releasing part of an expansion area upon request under any conditions. Nakhimovsky discloses a malloc stack pointer method that does not provide the functions recited in Claim 9.

Applicant's Claim 12 recites that the release of a "directly required area upon a reservation request coming from another individual system is dependent on the urgency of the respective reservation request". While Bishop discloses utilizing priority commands, the system of Bishop does not have directly required areas, much less expansion areas for a speculatively extended area. Again, Nakhimovsky simply uses a pointer and stack arrangement to allocate memory and does not ask one individual system to release memory to another individual system.

Applicant's Claim 16 recites that "upon a competing reservation request from a second said individual system, the reservation means reserve at least part of the speculatively extended area that is reserved for the first individual system for the second individual system". As discussed above, Bishop does not include speculatively extended areas for the threads, but instead utilizes a stack system wherein available memory is simply released and not related to a second individual system.

For the above reasons, Claims 6-9, 12 and 16 distinguish Nakhimovsky in view of Bishop and allowance thereof is respectfully requested.

Claims 20-25 have been cancelled in favor of new Claims 26-32. Claim 26 recites a method for regulating access to data in a data storage device that includes providing the data storage device with a plurality of speculatively extended

areas, each said speculatively extended area defined as a continuous block of memory in the data storage device corresponding to one of said individual systems. As discussed above, Nakhimovsky and Bishop both utilize a stack system and do not provide a continuous block of memory for the individual systems. Further Nakhimovsky and Bishop do not include "at least one directly required area and expansion areas for reserving at least one of free data and address areas" as recited in Claim 26.

Claim 26 further recites "providing an address statement from the one said individual system for requesting access to the reserved area of the speculatively extending area in the data storage device, the address statement comprising a first address defining the location in the reserved area to be released and a length statement defining a predetermined portion of the reserved area to be released". As discussed above, the applied prior art does not disclose providing an address statement from an individual system that requests access to a specific address in the data storage device and a specific length statement indicating the portion of the reserved area to be released.

Dependent Claims 27-32 are allowable for the reasons set forth above with respect to Claim 26 and include additional features that distinguish the prior art. For example, Claim 28 recites "reserving a second directly required area and at least a portion of a second expansion area of a second said speculatively extended area for a second one of said individual systems". Claim 28 further recites that the "step of reducing or increasing the size of the reserved area of said first speculatively extended area that is reserved for the first individual system comprises proportionally increasing or reducing the size of the second reserved area of the second individual system to utilize the full extent of the data storage device as a continuous block of memory". As discussed above, the applied prior art uses a stack and
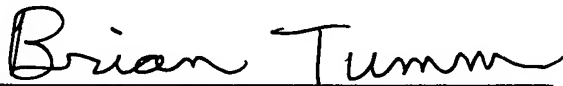
pointer method and does not provide the data as a continuous block of memory for the individual system.

For the above reasons, consideration and allowance of Claim 26, and Claims 27-32 dependent therefrom, is respectfully requested.

In light of the foregoing amendment and remarks, the claims remaining in the application should be considered in condition for allowance and early notice of allowability is courteously solicited.

Respectfully submitted,

Brian R. Tumm

BRT/ad

FLYNN, THIEL, BOUTELL & TANIS, P.C.	David G. Boutell	Reg. No. 25 072
2026 Rambling Road	Terryence F. Chapman	Reg. No. 32 549
Kalamazoo, MI 49008-1631	Mark L. Maki	Reg. No. 36 589
Phone: (269) 381-1156	Liane L. Churney	Reg. No. 40 694
Fax: (269) 381-5465	Brian R. Tumm	Reg. No. 36 328
	Donald J. Wallace	Reg. No. 43 977
	Stephen C. Holwerda	Reg. No. 57 391
	Dale H. Thiel	Reg. No. 24 323
	Sidney B. Williams, Jr.	Reg. No. 24 949
	Heon Jekal	Reg. No. L0379*
	*limited recognition number	

Encl: Wikipedia entry for "malloc" – 7 pages
      Postal Card

136.07/05

{00098303.DOC}

malloc

From Wikipedia, the free encyclopedia

In computing, `malloc` is a subroutine provided in the C programming language's and C++ programming language's standard library for performing dynamic memory allocation.

Contents

Rationale

The C programming language manages memory either *statically* or *automatically*. Static-duration variables are allocated in main (fixed) memory and persist for the lifetime of the program; automatic-duration variables are allocated on the stack and come and go as functions are called and return. For static-duration and, until C99 which allows variable-sized arrays, automatic-duration variables, the size of the allocation must be a compile-time constant. If the required size will not be known until run-time — for example, if data of arbitrary size is being read from the user or from a disk file — using fixed-size data objects is inadequate. Some platforms provide the `alloca` function,[1] which allows run-time allocation of variable-sized automatic variables on the stack.

The lifetime of allocated memory is also a concern. Neither static- nor automatic-duration memory is adequate for all situations. Stack-allocated data cannot persist across multiple function calls, while static data persists for the life of the program whether it is needed or not. In many situations the programmer requires greater flexibility in managing the lifetime of allocated memory.

These limitations are avoided by using dynamic memory allocation in which memory is more explicitly but more flexibly managed, typically by allocating it from a *heap*, an area of memory structured for this purpose. In C, one uses the library function `malloc` to allocate a block of memory on the heap. The program accesses this block of memory via a pointer which `malloc` returns. When the memory is no longer needed, the pointer is passed to `free` which deallocates the memory so that it can be used for other purposes.

Dynamic memory allocation in C

The `malloc` function is the basic function used to allocate memory on the heap in C. Its function prototype is

```
void *malloc(size_t size);
```

which allocates `size` bytes of memory. If the allocation succeeds, a pointer to the block of memory is returned.

`malloc` returns a void pointer (`void *`), which indicates that it is a pointer to a region of unknown data type. Note that because `malloc` returns a void pointer, it needn't be explicitly cast to a more specific pointer type: ANSI C defines an implicit coercion between the void pointer type and other pointer types. An explicit cast of `malloc`'s return value is sometimes performed because `malloc` originally returned a `char *`, but this cast is unnecessary in modern C code.[2][3] Omitting the cast creates an incompatibility with C++, which requires it.

Memory allocated via `malloc` is persistent: it will continue to exist until the program terminates or the memory is explicitly deallocated by the programmer (that is, the block is said to be "freed"). This is achieved by use of the `free` function. Its prototype is

```
void free(void *pointer);
```

which releases the block of memory pointed to by `pointer`. `pointer` must have been previously returned by `malloc` or `calloc` (or a function which uses one of these, eg `strdup`), and must only be passed to `free` once.

Usage example

The standard method of creating an array of ten integers on the stack is:

```
int array[10];
```

To allocate a similar array dynamically, the following code could be used:

```
/* Allocate space for an array with ten elements of type int. */
int *ptr = malloc(10 * sizeof (int));
if (ptr == NULL) {
    /* Memory could not be allocated, the program should handle the error here as appropriate. */
}
/* If ptr is not NULL, allocation succeeded. */
```

`malloc` returns a null pointer (NULL) to indicate that no memory is available, or that some other error occurred which prevented memory being allocated.

Related functions

calloc

`malloc` returns a block of memory that is allocated for the programmer to use, but is uninitialized. The memory is usually initialized by hand if necessary -- either via the `memset` function, or by one or more assignment statements that dereference the pointer. An alternative is to use the `calloc` function, which allocates memory and then initializes it. Its prototype is

```
void *calloc(size_t nelements, size_t bytes);
```

which allocates a region of memory large enough to hold `nelements` of size `bytes` each. The allocated region is initialized to zero.

realloc

It is often useful to be able to grow or shrink a block of memory. This can be done using `realloc` which returns a pointer to a memory region of the specified size, which contains the same data as the old region pointed to by `ptr` (truncated to the minimum of the old and new sizes). If `realloc` is unable to resize the memory region in-place, it allocates new storage, copies the required data, and frees the old pointer. If this allocation fails, `realloc` maintains the original pointer unaltered, and returns the null pointer value. The newly allocated region of memory is uninitialized (its contents are not predictable). The function prototype is

```
void *realloc(void *pointer, size_t size);
```

Common errors

The improper use of `malloc` and related functions can frequently be a source of bugs.

Allocation failure

`malloc` is not guaranteed to succeed — if there is no memory available, or if the program has exceeded the amount of memory it is allowed to reference, `malloc` will return a NULL pointer. Depending on the nature of the underlying environment, this may or may not be a likely occurrence. Many programs do not check for `malloc` failure. Such a program would attempt to use the NULL pointer returned by `malloc` as if it pointed to allocated memory, and the program would crash. This has traditionally been considered an incorrect design, although it remains common, as memory allocation failures only occur rarely in most situations, and the program frequently can do nothing better than to exit anyway. Checking for allocation failure is more important when implementing libraries — since the library might be used in low-memory environments, it is usually considered good practice to return memory allocation failures to the program using the library and allow it to choose whether to attempt to handle the error.

Memory leaks

When a call to `malloc`, `calloc` or `realloc` succeeds, the return value of the call should eventually be passed to the `free` function. This releases the allocated memory, allowing it to be reused to satisfy other memory allocation requests. If this is not done, the allocated memory will not be released until the process exits — in other words, a memory leak will occur. Typically, memory leaks are caused by losing track of pointers, for example not using a temporary pointer for the return value of `realloc`, which may lead to the original pointer being overwritten with NULL, for example:

```
void *ptr;
size_t size = BUFSIZ;

ptr = malloc(size);

/* some further execution happens here... */

/* now the buffer size needs to be doubled */
if (size > SIZE_MAX / 2) {
    /* handle overflow error */
    /* ... */
    return (1);
}
size *= 2;
ptr = realloc(ptr, size);
if (ptr == NULL) {
    /* the realloc failed (it returned NULL), but the original address in ptr has been lost
       so the memory cannot be freed and a leak has occurred */
    /* ... */
    return (1);
}
/* ... */
```

Use after free

After a pointer has been passed to `free`, it becomes a dangling pointer: it references a region of memory with undefined content, which may not be available for use. However, nothing prevents the pointer from being used. For example:

```
int *ptr = malloc(sizeof (int));
free(ptr);
*ptr = 0; /* Undefined behavior! */
```

Code like this has undefined behavior: its effect may vary. Commonly, the system may have reused freed memory for other purposes. Therefore, writing through a pointer to a deallocated region of memory may result in overwriting another piece of data somewhere else in the program. Depending on what data is overwritten, this may result in data corruption or cause the program to crash at a later time. A particularly bad example of this problem is if the same pointer is passed to `free` twice, known as a *double free*. To avoid this, some programmers set pointers to NULL after passing them to `free`: `free(NULL)` is safe (it does nothing).[4] However, this will not protect other aliases to the same pointer from being doubly freed.

Freeing unallocated memory

Another problem is when `free` is passed an address that wasn't allocated by `malloc`. This can be caused when a pointer to a literal string or the name of a declared array is passed to `free`, for example:

```
char *msg = "Default message";
int tbl[100];
```

passing either of the above pointers to free will result in undefined behaviour.

Implementations

The implementation of memory management depends greatly upon operating system and architecture. Some operating systems supply an allocator for malloc, while others supply functions to control certain regions of data.

The same dynamic memory allocator is often used to implement both malloc and operator new in C++. Hence, we will call this the **allocator** rather than malloc. (However, note that it is never proper for C++ to treat malloc and new interchangeably. For example, free cannot be used to release memory that was allocated with new.[5])

Heap-based

Implementation of the allocator on IA-32 architectures is commonly done using the heap, or data segment. The allocator will usually expand and contract the heap to fulfill allocation requests.

The heap method suffers from a few inherent flaws, stemming entirely from fragmentation. Like any method of memory allocation, the heap will become fragmented; that is, there will be sections of used and unused memory in the allocated space on the heap. A good allocator will attempt to find an unused area of already allocated memory to use before resorting to expanding the heap. However, due to performance it can be impossible to use an allocator in a real time system and a memory pool must be deployed instead.

The major problem with this method is that the heap has only two significant attributes: base, or the beginning of the heap in virtual memory space; and length, or its size. The heap requires enough system memory to fill its entire length, and its base can never change. Thus, any large areas of unused memory are wasted. The heap can get "stuck" in this position if a small used segment exists at the end of the heap, which could waste any magnitude of address space, from a few megabytes to a few hundred.

The glibc allocator

The GNU C library (glibc) uses both brk and mmap on the Linux operating system. The brk system call will change the size of the heap to be larger or smaller as needed, while the mmap system call will be used when extremely large segments are allocated. The heap method suffers the same flaws as any other, while the mmap method may avert problems with huge buffers trapping a small allocation at the end after their expiration.

The mmap method has its own flaws: it always allocates a segment by mapping entire pages. Mapping even a single byte will use an entire page which is usually 4096 bytes. Although this is usually quite acceptable, many architectures provide large page support (4 MiB or 2 MiB with PAE on IA-32). The combination of this method with large pages can potentially waste vast amounts of memory. The advantage to the mmap method is that when the segment is freed, the memory is returned to the system immediately.

OpenBSD's `malloc`

OpenBSD's implementation of the `malloc` function makes use of `mmap`. For requests greater in size than one page, the entire allocation is retrieved using `mmap`; smaller sizes are assigned from memory pools maintained by `malloc` within a number of "bucket pages," also allocated with `mmap`. On a call to `free`, memory is released and unmapped from the process address space using `munmap`. This system is designed to improve security by taking advantage of the address space layout randomization and gap page features implemented as part of OpenBSD's `mmap` system call, and to detect use-after-free bugs—as a large memory allocation is completely unmapped after it is freed, further use causes a segmentation fault and termination of the program.

Hoard's `malloc`

The Hoard memory allocator is an allocator whose goal is scalable memory allocation performance. Like OpenBSD's allocator, Hoard uses `mmap` exclusively, but manages memory in chunks of 64K called superblocks. Hoard's heap is logically divided into a single global heap and a number of per-processor heaps. In addition, there is a thread-local cache that can hold a limited number of superblocks. By allocating only from superblocks on the local per-thread or per-processor heap, and moving mostly-empty superblocks to the global heap so they can be reused by other processors, Hoard keeps fragmentation low while achieving near linear scalability with the number of threads.

Allocation size limits

The largest possible memory block `malloc` can allocate depends on the host system, particularly the size of physical memory and the operating system implementation. Theoretically, the largest number should be the maximum value that can be held in a *size_t* type, which is an implementation-dependent unsigned integer representing the size of an area of memory. The maximum value is `(size_t) -1`, or the constant `SIZE_MAX` in the C99 standard. The C standards guarantee that a certain minimum (0x7FFF in C90, 0xFFFF in C99) for at least one object can be allocated.

See also

- Buffer overflow
- Memory debugger
- mprotect
- Page size
- Aard
- new (C++)

References

1. ^ libc manual (http://www.gnu.org/software/libc/manual/html_node/Variable-Size-Automatic.html) on gnu.org accessed at March 9, 2007
2. ^ comp.lang.c FAQ list · Question 7.7b (http://www.c-faq.com/malloc/mallocnocast.html) on C-FAQ accessed at March 9, 2007
3. ^ FAQ > Explanations of... > Casting malloc (http://faq.cprogramming.com/cgi-bin/smartfaq.cgi?id=1043284351&answer=1047673478) on Cprogramming.com accessed at March 9, 2007
4. ^ The Open Group Base Specifications Issue 6

(http://www.opengroup.org/onlinepubs/009695399/functions/free.html) on The Open Group accessed at March 9, 2007

5. ^ Can I free() pointers allocated with new? Can I delete pointers allocated with malloc()? (http://www.parashift.com/c++-faq-lite/freestore-mgmt.html#faq-16.3) on C++ FAQ LITE accessed at March 9, 2007

External links

- Definition of malloc in IEEE Std 1003.1 standard (http://www.opengroup.org/onlinepubs/009695399/functions/malloc.html)
- The design of the basis of the glibc allocator (http://gee.cs.oswego.edu/dl/html/malloc.html) by Doug Lea
- Simple Memory Allocation Algorithms (http://www.osdcom.info/content/view/31/39/) on OSDEV Community
- "Hoard: A Scalable Memory Allocator for Multithreaded Applications (http://www.cs.umass.edu/~emery/pubs/berger-asplos2000.pdf) " by Emery Berger
- "Scalable Lock-Free Dynamic Memory Allocation (http://www.research.ibm.com/people/m/michael/pldi-2004.pdf) " by Maged M. Michael
- "*Inside memory management* - The choices, tradeoffs, and implementations of dynamic allocation (http://www-106.ibm.com/developerworks/linux/library/l-memory/) " by Jonathan Bartlett
- Memory Reduction (GNOME) (http://live.gnome.org/MemoryReduction) wiki page with lots of information about fixing malloc
- *A Scalable Concurrent malloc Implementation for FreeBSD* (http://citeseer.comp.nus.edu.sg/749403.html) by Jason Evans
- "TCMalloc: Thread-Caching Malloc" (http://goog-perftools.sourceforge.net/doc/tcmalloc.html) , a high-performance malloc developed by Google

Retrieved from "http://en.wikipedia.org/wiki/Malloc"

Categories: Stdlib.h | Memory management | C programming language | Articles with example C code

---

- This page was last modified 19:50, 6 December 2007.
- All text is available under the terms of the GNU Free Documentation License. (See **Copyrights** for details.)
  Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a U.S. registered 501(c)(3) tax-deductible nonprofit charity.